Proc. High Performance Computing Symposium of the Advanced Simulation Technologies Conference (Society for Modeling Simulation International), Arlington, VA, April 18-22, 2004.

LBNL-55522

Graph-Theoretic Methods in Simulation Using SPARK

by

Edward F. Sowell
Dept. of Computer Science
California State University, Fullerton
Fullerton, California, 92834
sowell@fullerton.edu

Michael A. Moshier
Dept. of Mathematics and Computer Science
Chapman University
Orange, California 92866
moshier@chapman.edu

Philip Haves
Lawrence Berkeley National Laboratory
Berkeley, California 94720
phaves@lbl.gov

Dimitri Curtil
Lawrence Berkeley National Laboratory
Berkeley, California 94720
dcurtil@lbl.gov

April 2004

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of the Building Technologies Program of the US Department of Energy, under Contract No. DE-AC03-76SF00098.

Graph-theoretic Methods in Simulation Using SPARK

Edward F. Sowell
Dept. of Computer Science, California State University, Fullerton
Fullerton, California, 92834
sowell@fullerton.edu

Michael A. Moshier
Dept. of Mathematics and Computer Science, Chapman University
Orange, California 92866
moshier@chapman.edu

Philip Haves
Lawrence Berkeley National Laboratory
Berkeley, California 94720
phaves@lbl.gov

Dimitri Curtil
Lawrence Berkeley National Laboratory
Berkeley, California 94720
dcurtil@lbl.gov

Keywords: reduction, decomposition, differential-algebraic systems, graph theory, hierarchical, equation based

ABSTRACT

This paper deals with simulation modeling of nonlinear, deterministic, continuous systems. It describes how the Simulation Problem Analysis and Research Kernel (SPARK) uses the mathematical graph both to describe models of such systems, and to solve the embodied differential-algebraic equation systems (DAEs). Problems are described declaratively rather than algorithmically, with atomic objects representing individual equations and macro objects representing larger programming entities (submodels) in a smooth hierarchy. Internally, in a preprocessing step, graphs are used to represent the problem at the level of equations and variables rather than procedural, multi-equation blocks. Benefits obtained include models that are without predefined input and output sets, enhancing modeling flexibility and code reusability, and relieving the modeler from manual algorithm development. Moreover, graph algorithms are used for problem decomposition and reduction, greatly reducing solution time for wide classes of problems.

After describing the methodology the paper presents results of benchmark tests that quantify performance advantages relative to conventional methods. In a somewhat contrived nonlinear example we show $O(n)$ performance as opposed to $O(n^3)$ for conventional methods and $O(n^2)$ for sparse methods. Another more realistic example deals with the model of a complex air-conditioning system. Comparative results show the number of simultaneous equations was reduced by a factor of 4 and the execution time reduced by a factor of ~15-20 when compared to a popular simulation program for problems in this domain. Both of these examples offer good opportunities for decomposition and reduction. A third example treats two dimensional heat transfer in homogeneous media. In this case the lack of opportunity for decomposition and reduction results in a still-large and sparse iterative problem. However, even for this example a sparse option within SPARK yields solution speeds comparable to a custom coded solution using state-of-the-art sparse packages.

Finally, we describe recent extensions that allow developers to mix SPARK's graph-theoretic modeling paradigm with conventional procedural methods.

BACKGROUND

We are concerned here with enhanced modeling flexibility and solution efficiency of systems of nonlinear DAEs. Such equation systems, which are deterministic rather than over- or under-determined, arise in nearly all engineering and scientific analysis, but perhaps most notably in simulation of physical and biological systems. Rather than the conventional vector-matrix representation, the methodology discussed here relies upon the mathematical graph, i.e., a set of nodes connected by a set of edges, as the principal data structure for both model representation and solution. While this is not the first time graph theory has been used in connection with equation system solving, previous graph theory usage has been, most often, as an aid in sparse matrix manipulation. In the work reported here, graph-theoretical methods are applied *directly to the nonlinear equations*. Thus the graph, rather than the matrix, is the primary structure for storing the problem structure and data, and graph algorithms are employed to determine a solution sequence, operating directly on the nonlinear equations. Moreover, the model equations are stored *individually*, rather than packaged into modules at run time, and treated *as equations*, rather than as formulas with assignment (algorithms). Simultaneous equations manifest as cyclic graphs that are dealt with by iteration, but, importantly, the size of the iteration vector is reduced through use of a cutset reduction algorithm. Moreover, the problem graph is decomposed into strongly connected components, providing a very powerful divide-and-conquer mechanism. Since simulation is often applied to engineered system comprising discrete components interconnected in sparse networks, the equation sets produce sparse graphs that are often susceptible to such reduction methods. Equation inverses are found beforehand using symbolic manipulation (computer algebra) software.

The SPARK project at The Berkeley Lab is based on work carried out at the IBM Los Angeles Scientific Center in the early 1980s (Silverman, Jurovics et al. 1981; Sowell and others 1984; Sowell, Taghavi et al. 1984; Levy and Low 1988). The first implementation at The Berkeley Lab, called SPANK, was by Jeffery Anderson (Anderson 1986), working with Edward F. Sowell, Walter F. Buhl and others. The basic graph-theoretic ideas, e.g., matching on bipartite graphs and cutting digraphs, came directly from contributions to the IBM project by Hanoch Levy and David Low. There is a remarkable, yet coincidental, similarity between the IBM work and that of Edwards (Edwards 1982), which was going on at approximately the same time.

GRAPH-THEORETIC BASIS

The process that leads eventually to numerical solution begins with graph-theoretic analysis of the problem, carried out in several stages. The objective of the first step, matching, is to select an appropriate equation to calculate each problem variable. To accomplish this, equation objects and problem variables are represented as the two disjoint node sets of a bipartite graph (bigraph), Figure 1.

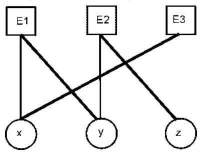

Figure 1 Matching

The upper nodes represent equations while the lower represent variables. Each edge incident upon an equation node indicates the existence of an inverse defined for the connected variable node in this equation. When viewed in this way, the selection of an equation for each problem variable is a matter of finding a complete matching on the graph, as indicated by the bold edges in the figure. There are several well-known bipartite matching algorithms (Hopcroft and Karp 1973; McHugh 1990). Upon completion of the matching there is a one-to-one relationship between equation objects and problem variables. Also, the matching identifies the particular inverse needed for each equation, e.g., E1 needs to be solved for $y = f_1(x)$ where $f_1(x)$ is an inverse of equation E1. The simulation environment includes symbolic tools for automatic generation of inverses for algebraic equations. If an inverse cannot be found for a particular variable in an equation, that edge can be omitted from the bigraph, thus preventing a matching requiring it. Since there are often many possible matchings for the bigraph, a suitable one is usually found, but if not an inverse can be specified in residual form, thus forcing the target variable to be a break variable (see below).

Subsequent processing steps require another graph to represent the problem. A directed graph (digraph) is formed in which there is a node for every problem variable. Keeping in mind that every variable is now associated with a particular equation, the selected equation is also associated with the node, e.g., E1 is associated with the *y* node in the above example. Each node can then be thought of as producing a value for the associated variable using the matched inverse equation. Directed edges are added to the graph showing dependency. Thus, Figure 2 derives from Figure 1.

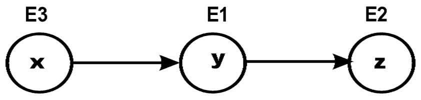

Figure 2 Digraph

The graph in Figure 2 is a simple, acyclic digraph that leads directly to a solution sequence; the nodes need only be "fired" in topological order[1]. More commonly, scientific problems involve equation sets that require simultaneous solution. Such problems produce graphs with closed circuits or cycles, e.g., Figure 3, requiring more graph-theoretic processing to get a solution sequence.

The graph in Figure 3 has an obvious property other than circuits, and that is strongly connected components, or *strong components* for short. These are maximal sets of nodes and edges in which every node can be reached from every other node. See (Aho, Hopcroft et al. 1983). These are circled with dashed lines in the figure.

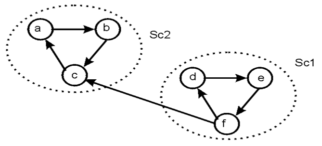

Figure 3 Cyclic digraph.

The importance of this characteristic in the current context is each strong component represents a subproblem that can be solved separately. Thus the strong component at the lower right, SC1, can be solved by itself, followed by solution of SC2. Discovering this structure in the problem graph is called *strong component decomposition*, for which there are well known algorithms, including one based on depth first search (Aho, Hopcroft et al. 1983). This algorithm is performed in SPARK as the first processing step of the digraph.[2] Fortuitously, the same algorithm also numbers the components in the order in which they must be solved, e.g., SC1 followed by SC2 in the example.

By the strict definition of a strong component, any node that is not in a cycle is itself a strong component. For example, Figure 2 has three strong components in this formal sense,

[1] Visit order in which all predecessors are visited first.

[2] Strong component decomposition is equivalent to reducing a matrix to block diagonal form.

although none have cycles. Acyclic groups of such degenerate strong components can also occur as parts of more complex graphs having cyclic strong components elsewhere. SPARK is able to recognize these structures and treat them as "acyclic components" for computational efficiency. That is, they are solved as a group, but sequentially rather than iteratively.

Once decomposed into components each cyclic component is processed with a *cutset algorithm*. A cutset is defined here as a set of nodes that, if removed, would break all cycles in the component. Cutsets are not unique. Indeed, the set of all nodes is a cutset, albeit a large one. While finding the minimum cutset is known to be an NP-complete problem, there are several good algorithms for finding relatively small cutsets (Levy and Low 1988).

The numerical solution sequence for a component with a non-zero cutset requires iteration. This is accomplished by "cutting" the graph at each cutset node. This can be viewed as inserting a new fictitious node to which the "from" end of all edges leaving the cut node are connected, thus leaving the cut node output disconnected. This is exemplified in Figure 4 where T is the cut node and T' is the fictitious node. This creates an acyclic graph that can be processed in topological order, just as in the Figure 2 example. Estimated values for the cut node variables are fed into the graph as inputs to the fictitious node and the graph is "fired," i.e., traversed in topological order executing the inverse associated with each node. In other words, the acyclic graph allows a residual to be calculated for each cut variable using the associated inverse. The cut digraph can thus be viewed as a vector-valued function of the input cutset, i.e., $\bar{x} = g(\bar{x}')$, which allows us to construct a conventional Newton-Raphson solution process, iterating until the change in each variable between successive iterations becomes smaller than the prescribed relative tolerance.

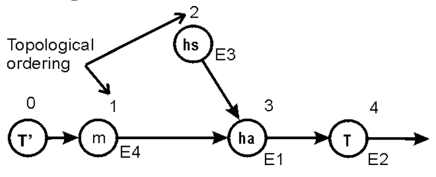

Figure 4 Cut digraph

Summarizing, the described graph-theoretic methodology comprises the following steps:

1. Construct and match a bipartite graph to identify an equation for each problem variable. Symbolic software is used to find the needed inverse of the matched equation.
2. Construct a digraph based on the matching.
3. Decomposition in to strong components, possibly acyclic as well as cyclic.
4. Find small cutset and construct N-R iterative problems for each cyclic component
5. Process all components in topological order.

HIERARCHICAL MODELING

Although SPARK operates at the equation level for graph-theoretic processing, and at run time, this level is not convenient for the modeler. Instead, a hierarchical language is provided. The basic modeling entities are *atomic classes* that represent individual equations, and *macro classes* that represent larger entities by incorporation of atomic classes and other macro classes. Both atomic and macro classes have *ports* that are linked together by the user with a simple network language to form larger models, such as macro classes and problems. This allows top down design in terms of models of major system components, followed by bottom up implementation. This hierarchy is flattened to equations and variables for processing as described in the previous section.

The user develops mathematical models for physical components in terms of equations, rather than algorithms. Each equation is written as an atomic class. C++ functions represent all feasible, explicit inverses of each equation, as well as those for which only residual form is possible; these inverses become part of the class. Atomic class ports include all variables that are present in the equation, with no distinction as to input or output. A symbolic tool can be used to automatically create an atomic class from an equation.

Macro classes, formed through composition from atomic and other macro classes, represent higher-level component models.[3] That is, the user defines a macro class by declaring instances of atomic classes and other macro classes, and linking them at their ports. Macro class creation is also automated with symbolic tools.

Both atomic classes and macro classes can be saved in libraries for reuse.

Because neither inputs nor outputs are designated in the class definition, we have *input/output free* or *multilateral* models. That is, the direction of computation is arbitrary. Being comprised of atomic classes, macro classes are also multilateral. As a result, which variables are inputs and which are calculated is not resolved until the overall *problem* is defined. At that time, the wanted input set is specified by a keyword in the network language. Thus, within the mathematical limits of requiring a well-posed problem, one does not have to redesign the entire model in order to change from solving for *x* given *p* to solving for *p* given *x*.

[3] SPARK is based on classes and objects, but does not support conventional OOP inheritance.

DYNAMIC MODELS

What we have described above is a modeling language and solution strategy for algebraic problems. However, the ideas are readily extended to dynamic problems. The most straightforward extension is to introduce a new problem variable for the derivative $\dot{x}$ of every dynamic variable *x*, and declare an integrator object that relates *x* and $\dot{x}$. Integrator objects are instances of classes that access saved histories of the dynamic variable, and employ them according to well-known numerical integration formulas, open or closed. This approach was described by Sowell and Buhl (Sowell and Buhl 1988). More recently, more elaborate built-in integration methods have been added to SPARK in order to afford proper error control. For example, *VisualSPARK 2* implements the Euler and the Trapezoidal integration methods with support for local error control through adaptive time stepping. These are described in the SPARK User's Manual (Sowell 1998), and more recent online documents (http://simulationresearch.lbl.gov).

IMPLEMENTATION

In the current SPARK implementation, the problem definition is first processed by the *parser*, scattering all macro objects into atomic objects and generating a flat problem representation in a file. All graph-theoretic processing is done in a second step called *setup*. The *setup* program emits a C++ source file in which the entire problem solution sequence, and all parameters, is embedded in data structures. The highest-level structure in this file is an instance of a C++ class called Problem. The Problem class has an array of Component objects, each of which contains the solution sequence for one of the strong components of the graph. This sequence is basically a list of pointers to functions representing the selected atomic object inverses at each node. The file is compiled, as are any new atomic classes, and everything is linked with a fixed library (or at run time with a dynamic link library) containing the fixed elements of the solver. The output of the link step is an executable that solves the problem numerically by stepping through the list of components, and through the solution sequence within each component. A component with cycles is converged with Newton-Raphson iteration on the cutset vector. There is a time loop to repeat the process if the problem is a dynamic one.

There are several other ways the concepts described previously could be implemented. We have experimented with interpretive implementations, eliminating the need for problem compilation and linkage entirely. Recently, a "dynamic" building mechanism has been added to *VisualSPARK 2* in which the static loading mechanism based on the C++ source file is completed at runtime. This technique allows SPARK to operate without a C++ compiler and linker, as long as all atomic classes are compiled beforehand and placed in DLLs. This approach improves productivity during model development with a modest reduction in runtime efficiency.

SYMBOLICS

A basic assumption in the methodology is the availability of at least one, and preferably more, inverses for the equation represented by each atomic class. This is needed for the matching process to succeed. Obtaining explicit inverses for the equations usually requires no more than high school algebra, but still can be tedious if there are many classes to be developed. Fortunately, computer algebra tools, e.g., *Maple* and *MACSYMA*, are now commonly available to aid in this task. SPARK comes with a free symbolic tool called *sparksym*, based on a licensed shareware program called *Mathomatic*, that allows the user to automatically construct not only the inverses but also the entire atomic class based on an entered equation. Macro classes can also be constructed by *sparksym*. Although the *Mathomatic*-based tool is limited to relatively simple equations, it may be sufficient for many users. If more powerful tools are needed, either *Maple* or *MACSYMA* can be installed and optionally used as the symbolic engine in *sparksym*.

USER INTERFACES

SPARK can be executed at the command line, or through graphical user interfaces. At the command line one uses a conventional text editor to create class and problem files. Then a single command carries out the parsing, setup, compilation and linkage steps. The symbolic tools can also be executed at the command line to automate creation of new classes. Graphical user interfaces, on the other hand, provide the user with an integrated development environment (IDE) in which problems and classes can be created, built and executed. The symbolic tools are also available within the IDE.

There are two such IDEs, one that comes with *VisualSPARK* released by The Berkeley Laboratory (http://simulationresearch.lbl.gov), and another that comes with *WinSPARK* released by Ayres Sowell Associates, Inc. (http://www.ayressowell.com). The two interfaces reflect different developmental heritages, with *WinSPARK* exhibiting more of the look and feel of Microsoft Windows, and *VisualSPARK* that of Unix, although the latter has been ported to Windows. Ostensibly, both packages have the same SPARK core, but in practice the two products sometimes get out of synchronization, with differences among newer features. Documentation for both products is available online.

PERFORMANCE BENCHMARKS

In addition to the advantages already noted, e.g., a declarative hierarchical modeling style and multilateral object-based models, the graph-theoretic modeling paradigm also offers a significant performance advantage over conventional and sparse methods for some problem classes. In our benchmarking work we have attempted to show asymptotic performance of the methodology, as implemented in SPARK, and to compare it to the more commonly used sparse matrix methods and other simulation software. Like any such effort, our study was limited by time and resource constraints. With these constraints in mind, we chose problems representative of both those well suited and those poorly suited to the methodology, as well as one real-world problem for which the performance of another simulation program was available from earlier work. Naturally, we used software conveniently available to us for comparison. While not exhaustive, we feel that these benchmarks are sufficient to show, at the least, the promise of the methodology. Here we only summarize these studies and results, which have been more fully reported elsewhere (Sowell and Haves 2001).

A Good Problem for the Methodology

The first benchmark problem was selected to represent problems well suited to the graph-theoretic methodology. It derives from a problem in the SPARK Users' Manual consisting of four highly nonlinear equations. For the study, this set of equations was implemented as a SPARK macro class, which was then instantiated *n/4* times to get a problem of size *n*. Obviously, every instance is then a separately solvable problem, although all are presented to SPARK as a single problem. SPARK is able to discern this structural regularity from the problem graph and partition it into *n/4* strongly connected components. It develops that each has a cutset of size 1. Consequently, during the numeric phase of the solution, *n/4* single-variable iterative solutions are carried out.

For comparison purposes, this equation set was also solved with three other methods. First, a Newton-Raphson nonlinear solver *nlsolve* was handcrafted for this equation set, with the problem size as an input parameter. In this solver, the four equations were coded in a single function that was called as needed for calculation of the residual functions and the Jacobian. The matrix functions from SPARK were used to calculate the Jacobian numerically and solve the linear set for new estimates of the iteration variables. Second, a sparse Newton-Raphson solver *spnlsolve* was written using the sparse LU solve function from the *Meschach* sparse matrix package (Stewart and Leyk 1994). Comparative run times are shown in Figure 5.

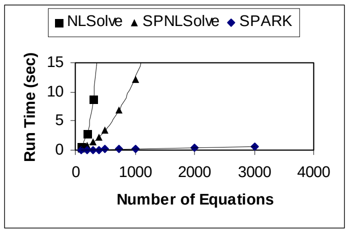

Figure 5 Comparative run times

As would be expected, the experimental results show $O(n^3)$ performance for the full matrix solution. The solver based on the *Meschach* sparse matrix functions shows much better performance, approximately $O(n^2)$. Also as expected, SPARK is much better than the sparse implementation, showing about $O(n)$. In order to confirm that these dramatic solution speed improvements are not attributable to the particular sparse package chosen, this problem was also coded for solution with the *SuperLU* sparse package (Demmel, Gilbert et al. 1999). If not the most advanced software in this category, it appears at least to be among the most current. The upshot of the results (Sowell and Haves 2001) is that although *SuperLU* is considerably faster than *Meschach* (15 seconds. versus 48 seconds for *n* = 2000), SPARK is still much faster. More importantly, it is clear that *SuperLU*, like *Meschach*, is performing at $O(n^2)$ as compared to $O(n)$ for SPARK.

A Poor Problem for the Methodology

The second benchmark problem, purposely chosen *not* to be well suited to the graph-theoretic methodology, is Laplace's equation in two dimensions. This equation, among other things, models heat transfer in a thin, square plate with a uniformly distributed heat source and uniform boundary temperature. The problem is discretized by dividing the square into a uniform grid of specified size. Each cell in the grid is represented by a nodal temperature $T_{i,j}$ and is governed by a heat balance equation

$$\begin{aligned} q_{si,j} &= (T_{i,j} - T_{i,j+1}) + (T_{i,j} - T_{i-1,j}) \\ &+ (T_{i,j} - T_{i,j-1}) + (T_{i,j} - T_{i+1,j}) \end{aligned}$$

where $q_{si,j}$ is the heat source rate per unit surface area. The internodal conductance is assumed to be 1. This problem was coded for solution with *Meschach* using sparse LU factorization.[4] For comparison, a program was written to generate SPARK problem and input files for the same equation system. The grid size was varied between 3 and 45, yielding equation set sizes between 9 and 2025. Both SPARK and the *Meschach*-based solver were compiled with the same compiler and optimization options. Also, care was taken to be sure that the same problem, in terms of equation count and math operations, was being solved in both cases. Then, because we knew this problem provided little opportunity for reduction or decomposition, the SPARK

[4]This choice was made since Cholesky factorization and sparse conjugate gradient iteration applies only to symmetric positive definite matrices, a condition satisfied by the Laplacian but not often found in general simulation problems

solver was modified to optionally use either sparse or non-sparse vector-matrix data structures and functions from *Meschach* when updating the solution vector. The results showed that while *Meschach* outperformed the standard version of SPARK, a version with sparse handling of the reduced Jacobian performed essentially the same as *Meschach* (Sowell and Haves 2001).[5]

A Real World Problem

Going beyond the rather contrived examples above, the graph-theoretic methodology was also evaluated by modeling an airflow system employing discrete-time controllers. The example used was a typical air-conditioning airflow network and its associated control loops, a problem involving significant computational burden (Haves, Norford et al. 1998). A number of steady state component models were implemented as SPARK objects, including variable speed centrifugal fans, flow diverters, flow mixers and control dampers. In modeling air flow, a square law dependence of total pressure drop on flow rate was used above a critical flow rate and a linear dependence was used below the critical flow rate to avoid known computational problems when the assumption of turbulent flow is applied to low flow rates. Dynamic models included flow sensors, pressure sensors, discrete-time proportional-plus-integral (PI) controllers and rate limits in fan speed controllers. In order to assess the benefits of using SPARK methods, a base case and two reference cases were constructed. The base case was modeled with SPARK in the normal manner, allowing the graph-theoretic techniques to perform reduction of the problem graph. The two reference cases were (a) The system modeled using the HVACSIM+ program (Clark 1985), and (b) the system modeled using SPARK, but inhibiting the normal problem reduction techniques. The use of the two reference cases enables the benefits of the graph-theoretic techniques to be separated from the effects of program architecture. For all three cases, the simulation problem was a series of set-point changes for each controller followed by a disturbance caused by progressive closing of the variable volume terminal boxes. In addition to these comparisons directed at assessment of the importance of reduction, a side study was performed to determine whether "breaking" of control loops offers computational advantage. The interest in this derives both from the need to model sample-and-hold in discrete-time controllers, and from the introduction of artificial delays as a computational device to speed solution. Results are shown in Table 1.

[5] The current version of *VisualSPARK* offers a sparse option.

Table 1Results for air-conditioning system

Control	Time (s)		Iteration Variables	
	HVAC SIM+	SPARK	HVAC SIM+	SPARK
Intact	1135	48.8	62	15
Broken	785	52.7	55	15

In the first comparison, "Control loops Intact," the flow network equations and the controller equations are solved simultaneously. The main result is that SPARK is 15 - 20 times faster than HVACSIM+. The obvious reason for the speedup is that SPARK achieves a *4:1* reduction in the number of variables in the iteration vector relative to the conventional program. In the second comparison, "Control loops Broken," the set of simultaneous equations representing the airflow network and those representing the control system are solved sequentially. This corresponds to breaking the algebraic loops, e.g., by introduction of a sample-and-hold in the controller or an artificial delay. Whereas a significant benefit was gained from breaking the control loops when using HVACSIM+, there was no such benefit when using SPARK. The reason for this, as discussed in another paper (Haves and Sowell 1998), is that SPARK finds two break variables that not only break the two control loops but also break other computation loops, so the control loops are broken regardless.

Table 1 shows results for a particular flow network. Experiments with a number of different flow network configurations show that SPARK achieves similar reductions in problem size across a range of different network topologies typical of fluid distribution systems. This suggests that the application of the problem reduction methods used in SPARK to such problems will result in increases in computational efficiency similar to those reported here.

DISCUSSION

The above results confirm that the graph-theoretic methodology offers significant reduction in solution times relative to both conventional and sparse matrix methods in the solution of certain kinds of nonlinear equation systems. This is borne out most dramatically by the contrived nonlinear benchmark problem, but is also quite clear from the air-conditioning control application. However, in the case of the example involving Laplace's equation, we observe that without using sparse methods itself, SPARK has difficulty competing with sparse solvers. Understanding why this occurs is important in order to guide improvement of the graph-theoretic methods, and to delineate properly the class of problems where one should expect the most dramatic improvements with them.

To understand the observed differences in run times, it is important to note that at the heart of the Newton-Raphson nonlinear solution process is the solution of a linear system in order to get the correction vector for iteration variables.

In general, this is an $O(n^3)$ process. Even with sparse methods it is typically $O(n^2)$. Consequently, anything that can be done to reduce the size of the iteration vector has a powerful effect, especially for large problem size. SPARK does exactly this in two separate ways: strong component decomposition and cutset reduction. Decomposition is possible when the equation set turns out to be a sequence of separately solvable problems. SPARK routinely detects this property and carries out the decomposition without user intervention or custom coding. For example, the nonlinear benchmark problem with *100* equations and variables is decomposed into *25* subproblems each of size *4*. This alone would reduce the run time by a factor of *625*. Cutset reduction refers to reducing the sizes of the iteration vectors (and Jacobians) within the subproblems. Since a cutset of size 1 was discovered in each component of our example, the iteration vectors are all of size 1, so the overall theoretical run time reduction is by a factor of *40,000*. Of course, the theoretical efficiency gain is only partially realized due to overhead, but this analysis clearly explains the observed excellent performance for the Good and Real examples above. It also explains why we should expect little or no advantage for the Poor example. Highly interconnected problems like this one offer no opportunity for decomposition, and make cutset reduction much more difficult. But since many real simulation problems have portions that are poor in this sense, as well as sparsely connected portions that are ideal for graph-theoretic methods, the challenge is to be sure that the graph-theoretic implementation at least *does no worse* than conventional sparse methods in the Poor parts of the problem.

It should also be noted that while reduction to block diagonal form, which is equivalent to strong component decomposition, can be done with sparse matrix packages, and is indeed done in some nonlinear equation solvers (Klein 1991), one should not assume that it is routinely done by all sparse solvers. Moreover, we are aware of no nonlinear solver other than SPARK that routinely does iteration vector reduction by use of small cutsets, in spite of the fact that the concept is well known, at least in the theoretical sparse analysis community. The reasons for this apparent lack of widespread use of these important techniques may be complications attendant to the vector-matrix problem formulation commonly used in nonlinear solvers. We know for certain, on the other hand, that when the nonlinear problem is represented directly as graphs at the equation-variable level, routine application of the techniques is more or less intuitive, and the algorithms are straightforward.

MIXED PARADIGMS

SPARK is only one example of how the graph-theoretic modeling paradigm can be implemented. Many others applications of the paradigm are conceivable, including "mixed paradigm" projects in which graph-theoretic methods are used in some places while procedural methods are used in others. To enable such usage, recently Ayres Sowell Associates has implemented modifications to make SPARK internal methods directly accessible to model developers in non-SPARK environments. With one feature, called *SPARK Model Functions (SMF)*, developers can create SPARK system models of arbitrary size and complexity that can be called as ordinary functions by foreign executive programs. *WinSPARK* menus provide tools that automatically generate SMFs, placing them in DLLs for easy runtime access by other software. As a demonstration of possible usage, we have created a DLL that integrates with Microsoft Excel, thus providing SPARK generated models as custom user functions in this ubiquitous application (Sowell and Moshier 2003). Another example of using SPARK as a calculation kernel provides a link between SPARK and the *EnergyPlus* program. Individual *EnergyPlus* components are implemented as SPARK problems, and each such problem is instantiated and invoked at runtime on demand using the atomic class DLLs. (See SPARK Problem Driver API documentation at http://simulationresearch.lbl.gov/)

Another feature, called *Multi-Value Objects (MVOs)*, allows a developer to use a foreign procedural function within a SPARK model. This is useful when the there is a legacy model written in perhaps *FORTRAN*, *C*, or *C++* and time or other factors argue against re-implementation as an equation-based SPARK macro class. MVOs are also very useful when there is a set of equations within a system that are numerically problematic for a global solver, but which can be reliably solved simultaneously with well-known procedural algorithms. In both situations, there are multiple equations being solved for multiple variables simultaneously within the procedural model, in contrast to the SPARK policy of working with the individual equations and variables, solved globally. In *WinSPARK*, the MVO is treated as a macro object when developing the SPARK problem definition, but at run time the procedural function is invoked, returning any number of values to be propagated through the problem graph exactly like values calculated in the normal manner. *WinSPARK* offers symbolic tools to automatically generate MVOs. An alternative MVO implementation is used in *VisualSPARK.*

CONCLUSIONS

The principle conclusion that can be drawn from this work is that graph-theoretic techniques, as implemented in SPARK, can outperform conventional and sparse matrix methods for solution of problems that can be decomposed and/or reduced. Roughly speaking, execution time savings will be $O(mr^3)$ where *r* is the ratio of the largest strong component cutset size to the number of equations in the problem, and *m* is the number of strongly connected components into which the problem partitions. The reduction techniques produced close to the maximum reduction in the benchmark air-conditioning problem, and there are indications that similar reductions can be expected in the broad class of problems involving flow networks and their associated control systems. Reductions in execution time of more than an order of magnitude can be expected relative to full-matrix solvers such as HVACSIM+. While direct benchmarks were not carried out for simulators of flow networks that use sparse solvers, our indirect tests suggest that the sparse methods employed in such programs will not be comparable to SPARK for problems in this class. This is because sparse packages such as *SuperLU* do not *automatically* perform decomposition or reduction, notwithstanding the fact that handcrafted solvers based on sparse methods should be able to do so. On the other hand, problems characterized by a high degree of interconnectivity, such as energy, mass, or momentum transport in homogenous media, allow very little reduction and therefore are not *prima fascia* candidates for graph-theoretic solution methods. However, since the reduced Jacobian in homogeneous transport problems is still very sparse, conventional sparse matrix methods can be beneficially applied after decomposition and reduction. When this is done, the graph-theoretic solution method can be competitive with sparse solvers even for homogeneous transport problems, and probably superior for system simulations in which reducible *and* homogeneous transport components must both be solved.

In addition to performance improvements, the graph-theoretic paradigm allows convenient hierarchical, equation-based modeling. Additional benefits include input/output free models, meaning that the model does not have to be altered just because the input set changes, and relieving the modeler from the burden of algorithm development.

Finally, we call attention to several ways that the graph-theoretic methodology used in SPARK can be incorporated in other modeling environments. In particular, SPARK models can be incorporated into non-SPARK environments, and *vice versa*.

ACKNOWLEDGEMENTS

This work has been sponsored in part by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of the Building Technologies Program of the U. S. Department of Energy.

REFERENCES

Aho, A. V., J. E. Hopcroft, et al. (1983). *Data Structures and Algorithms*. Reading, MA, Addison Wesley.

Anderson, J. L. (1986). *A Network Language for Definition and Solution of Simulation Problems*, Lawrence Berkeley Laboratory

Clark, D. R. (1985). *HVACSIM+ Building System and Equipment Simulation Program (Reference Manual and User's Guide)*, National Institute of Standards and Technology (NIST)

Demmel, J. W., J. R. Gilbert, et al. (1999). *SuperLU User's Guide*. Berkeley, CA, University of California, Dept. of Computer Science

Edwards, D. W. (1982). *Robust Decomposition Techniques for Process Design and Optimization*. Chemical Engineering. London, University of London**:** 243

Haves, P., L. K. Norford, et al. (1998). "A Standard Simulation Testbed for Evaluation of Control Algorithms & Strategies." *Transactions of the American Society of Heating, Refrigerating, and Air-conditioning Engineers* **104**(1).

Haves, P. and E. F. Sowell (1998). "The Application of Problem Reduction Techniques Based on Graph Theory to the Simulation of Nonlinear Continuous Systems". *EuroSim*, Manchester, England, Society For Computer Simulation.

Hopcroft, J. and R. Karp (1973). "A n^5/2 Algorithm for Matching on a Bipartite Graph." *SIAM J. Algorithms.*

Klein, S. (1991). *Engineering Equation Solver (EES).* Madison, F-Chart Software

Levy, H. and D. W. Low (1988). "Contraction Algorithm for Finding Small Cycle Cut Sets." *J. Algorithms* **9**: 470-493.

McHugh, J. (1990). *Algorithmic Graph Theory*. Englewood Cliffs N.J. 07632, Prentice Hall.

Silverman, G., S. A. Jurovics, et al. (1981). "Modeling and Optimization of HVAC Systems Using Network Concepts." *ASHRAE Trans.* **87**(Pt. 2): 585-597.

Sowell, E. F. (1998). *SPARK Users' Manual*. Placentia, CA 92871, Ayres Sowell Associates, Inc.

Sowell, E. F. and W. F. Buhl (1988). "Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK)". *Proceedings of the USER-1 Building Simulation Conference*, Ostend, Belgium, Soc. for Computer Simulation International.

Sowell, E. F. and P. Haves (2001). "Efficient Solution Strategies for Building Energy System Simulation." *Energy and Buildings* **33**(4): 291-415.

Sowell, E. F. and M. A. Moshier (2003). "Application of the SPARK Kernel". *Building Simulation '03*, Eindhoven, International Building Performance Simulation Association.

Sowell, E. F. and others (1984). "Conventional Control Models for HVAC Network Simulation". *Proceedings of the Workshop on HVAC Controls Modeling and Simulation*, Georgia Institute of Technology.

Sowell, E. F., K. Taghavi, et al. (1984). "Generation of Building Energy System Models." *ASHRAE Trans.* **90**(Pt. 1): 573-86.

Stewart, D. E. and Z. Leyk (1994). "Meschach: Matrix Computation in C". *The Centre for Mathematics and Its Applications*, The Australian National University.